Programmation et modélisation orientées objet
(USAL5A & USAL2H)

Tests unitaires en Java

1 Les tests unitaires avec jUnit 5

Les principales assertions pour jUnit 5 sont rappelées dans le document suivant :

https://cedric.cnam.fr/sys/crolard/enseignement/GLG101/jUnit.pdf

La documentation officielle de jUnit 5 est diponible ici :

https://junit.org/junit5/docs/current/user-guide

Exercice

1. Définir la fonction suivante et la tester en utilisant jUnit 5.

```
public int add(int x, int y) {
   return x + y;
}
```

2. Définir la fonction suivante et la tester (en testant aussi le cas de la division par zéro).

```
public int div(int x, int y) {
   return x / y;
}
```

3. Modifier le code de la fonction `div` de manière à rendre le cas de la division par zero explicite dans le code et tester à nouveau.

4. Définir la fonction suivante et la tester :

```
public int prod(int x, int y) {
   boolean zero = false;
   if (x == 0)
      zero = true;
   if (y == 0)
      zero = true;
   if (zero)
      return 0;
   else
      return x * y;
}
```

2 Tests parametrisés

2.1 Format CSV

Depuis la version 5 de jUnit, il est possible de regrouper des cas de test en utilisant les annotations montrées sur l'exemple suivant :

```java
@ParameterizedTest
@CsvSource({"0, 1, 0", "1, 0, 0", "1, 1, 1"})
void testProd(int x, int y, int expected) {
   int actual = calc.prod(x, y);
   assertEquals(expected, actual);
}
```

La syntaxe utilisée est le format CSV (*Comma Separated Values*), avec une chaîne de caractères par cas de test. Le nombre et l'ordre des arguments donnés doit correspondre à ceux de la fonction de test. Des heuristiques sont utilisées pour reconnaître les valeurs des arguments et les convertir vers le type attendu.

Les détails sont disponibles dans la documentation de jUnit 5 :

https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests

Question

1. Reprendre les exercices du TP précédent, et regrouper les différentes données de test de chaque fonction en utilisant le format CSV.

2.2 Fichier CSV externe

Créer un nouveau fichier `"testProd2.csv"` dans le répertoire `test`, et le remplir avec quelques données de test au format CSV. On pourra utiliser simple éditeur de texte, ou un tableur, pour saisir et éditer ces données de test, ou installer un plugin de l'IDE adapté.

Remarque. L'option `numLinesToSkip = 1` permet d'ajouter une ligne d'entête dans le fichier CSV, et elle sera alors ignorée par jUnit.

```java
@ParameterizedTest
@CsvFileSource(resources = "/testProd2.csv", numLinesToSkip = 1)
void testProd2(int x, int y, int expected) {
   int actual = calc.prod(x, y);
   assertEquals(expected, actual);
}
```

Question

1. Ecrire une fonction `shortest` qui prend en entrée trois `String` et renvoie la plus courte. Tester cette fonction en regroupant les données de test dans un fichier CSV externe.

2.3 Données de test « générées »

Il est aussi possible générer les cas de test, et de les stocker dans une collection, ou sous forme de `Stream`.

```java
static IntStream range() {
   return IntStream.range(0, 20).skip(10);
}

@ParameterizedTest
@MethodSource("range")
void testWithRangeMethodSource(int x) {
   int actual = calc.add(x, x);
   int expected = 2 * x;
   assertEquals(expected, actual);
}
```

Question

1. Proposer une assertion pour vérifier la fonction `shortest`.

3 Objets et Valeurs

Implanter la classe `Person` suivante :

```
public class Person {

        String firstName;
        String lastName;

        public Person(String firstName, String lastName) {
                this.firstName = firstName;
                this.lastName = lastName;
        }
}
```

Questions

1. Tester la méthode `toString` par défaut (héritée de la classe `Object`).
2. Depuis l'IDE, générer la méthode `toString` pour votre classe et la tester à nouveau.
3. Tester la méthode `equals` par défaut (héritée de la classe `Object`).
4. Depuis l'IDE, générer la méthode `equals` pour votre classe et la tester à nouveau.

A Projet Maven

Voici l'arborescence standard d'un projet Maven :

Répertoire	**Contenu**
`/src/main`	les fichiers sources principaux
`/src/main/java`	le code source (compilé dans `/target/classes`)
`/src/main/resources`	les fichiers de ressources (à la compilation, le contenu de ce répertoire est copié dans `/target/classes` puis inclus dans l'artéfact généré)
`/src/test/java`	le code source des tests (compilé dans `/target/test-classes`)
`/src/test/resources`	les fichiers de ressources pour les tests
`/target/classes`	les classes compilées
`/target/test-classes`	les classes compilées des tests unitaires
`/pom.xml`	le fichier POM de description du projet